DOCUMENT PROCESSING GUIDE

UNIX SYSTEM

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS PAGE

CONTENTS

I. INTRODUCTION

An important feature of the UNIX operating system is to provide a method of document preparation and generation. The Document Preparation section (Section II) contains three parts:

- Advanced Editing
- Stream Editor
- Miscellaneous Facilities.

The Advanced Editing part describes text editing functions which include special characters, line addressing, global commands, commands for cut and paste operations, and text editor-based programs. The Stream Editor part describes the noninteractive context editor, and the Miscellaneous Facilities part outlines some other facilities that aid in document preparation.

The Formatting Facilities section (Section III) contains three parts:

- NROFF and TROFF User's Manual
- Table Formatting Program
- Mathematics Typesetting Program.

The NROFF and TROFF User's Manual part presents information to enable the user to do simple formatting tasks and to make incremental changes to existing packages of **troff** formatter commands. The UNIX operating system formatter, **nroff**, is identical to the **troff** formatter in most respects. The Table Formatting Program part describes the **tbl** program and the input commands used to generate documents that contain tables. The Mathematics Typesetting Program part describes the program usage and language for obtaining text with mathematical expressions. The language interfaces directly with the **troff** processor so mathematical expressions can be embedded in the running text of a manuscript and the entire document produced in one process.

The Memorandum Macros section (Section IV) is a user's guide and reference manual for the Memorandum Macros (MM). These macros provide a general purpose package of text formatting macros used with the **nroff** and **troff** formatters. The macros provide users of the UNIX operating system a unified, consistent, and flexible tool for producing many common types of documents. Although the UNIX operating system provides other macro packages for various specialized formats, MM is the standard general purpose macro package for most documents such as letters, reports, technical memoranda, released papers, manuals, books, design proposals, and user guides. Uses of MM range from single-page letters to documents of several hundred pages in length.

The Viewgraphs and Slides Macros section (Section V) describes the MV package of macros. These macros provide users a method of preparing viewgraphs and slides using the **troff** formatter. Included is a discussion on the use of the macros and a philosophy of how a viewgraph or slide should appear.

II. DOCUMENT PREPARATION

ADVANCED EDITING

1. Introduction

The advanced editing part is meant to help UNIX operating system users (secretaries, typists, programmers, etc.) make effective use of facilities for preparing and editing documents, text, programs, files, etc. It provides explanations and examples of:

- special characters, line addressing, and global commands in the text editor (**ed**)
- commands for "cut and paste" operations on files and parts of files, including **mv**, **cp**, **cat**, and **rm** commands, and **r**, **w**, **m**, and **t** commands of the text editor
- editing scripts and text editor-based programs like **grep** and **sed**.

Although this document is written for nonprogrammers, new UNIX operating system users with any background should find helpful hints on how to get their jobs done more easily. The UNIX operating system provides effective tools for text editing, but that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, users who are not computer specialists (typists, secretaries, casual users) often use the UNIX operating system less effectively than they could. The reader should be familiar with the material in the "Basics For Beginners" section of the User's Guide—UNIX Operating System before using the text editor. Further information on all commands discussed here can be found in the User's Manual—UNIX Operating System.

Examples are based on experience and observations of users and the difficulties encountered. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions on the effective use of related tools, e.g., those for file manipulation and those based on **ed**.

The next paragraphs discuss shortcuts and labor-saving devices. Not all will be instantly useful (some will) and others should provide ideas for future use. Until these things are used to build confidence, they will remain theoretical knowledge.

> *Note:* A document like this should provide ideas about what to try. There is only one way to learn to use something, and that is to use it. Reading a description is no substitute for hands-on use.

2. Special Characters

The **ed** program is the primary interface to the system, so it is worthwhile to know how to get the most out of it with the least effort.

2.1 Print and List Commands

Two commands are provided for printing contents of lines being edited. Most users are familiar with the print command (**p**) in combinations like

```
1,$p
```

to print all lines that are being edited, or

```
s/abc/def/p
```

to change "abc" to "def" on the current line and to print the results. Less familiar is the list command (**l**) which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible.

such as tabs and backspaces. If a line listed contains some of these, l will print each tab as ">" and each backspace as "<". This makes it easier to correct typing mistakes that insert extra spaces adjacent to tabs or a backspace followed by a space.

The l command also "folds" long lines for printing. Any line that exceeds 72 characters is printed on multiple lines. Each printed line except the last is automatically terminated by a backslash (\) to indicate that the line was folded. A "$" character is appended to the real end of line. This is useful for printing long lines on terminals having output line capability of only 72 characters per line.

Occasionally, the l command will print in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally do not print, e.g., form feed, vertical tab, or bell. Each such combination is interpreted as a single character. When such characters are detected, they may have surprising meanings when printed on some terminals. Often their presence means that a finger slipped while typing.

2.2 Substitute Command

The substitute command (**s**) is used for changing the contents of individual lines. It is probably the most complex and effective of any **ed** command.

The meaning of a trailing global command after a substitute command is illustrated in the next two commands:

```
s/this/that/
```

and

```
s/this/that/g
```

The first form replaces the first "this" on the line with "that". If there is more than one occurrence of "this" on the line, the second form (with the trailing **g**) changes all of them. Either of the two forms of the **s** command can be followed by **p** or **l** to print or list the contents of the line:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

All are legal and have slightly different meanings.

An **s** command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines specified by the line numbers. Thus:

```
1,$s/mispell/misspell/
```

changes the first occurrence of "mispell" to "misspell" on every line of the file. The following command changes every occurrence in every line:

```
1,$s/mispell/misspell/g
```

By adding a **p** or **l** to the end of any of these substitute commands, only the last line that was changed will be printed, not all lines. How to print all the lines that were changed is described later.

Any character can be used to delimit pieces of an **s** command. There is nothing sacred about slashes (but slashes must be used for context searching). For instance, for a line that contains a lot of slashes already, e.g:

```
//exec //sys.fort.go //etc...
```

a colon could be used as the delimiter. To delete all the slashes, the command is

```
s/:/::g
```

2.3 Undo Command

Occasionally, an erroneous substitution will be made in a line. The undo command (**u**) negates the last command so that data is restored to its previous state. This command is useful after executing a global command if it is discovered the command did things that are undesirable.

2.4 Metacharacters

When using **ed**, certain characters have unexpected meanings when they occur in the left side of a substitute command or in a search for a particular line. These are called "metacharacters" which are:

- Period .
- Backslash \
- Dollar Sign $
- Circumflex ^
- Star *
- Brackets [ ]
- Ampersand. &

Even though metacharacters are discussed separately in the following text, they can be combined. An example is given in the paragraph on "Circumflex" (2.4.4).

2.4.1 Period

The period (.) on the left side of a substitute command or in a search with /.../, stands for any single character. Thus the search

```
/x.y/
```

finds any line where "x" and "y" occur separated by a single character, as in

```
x+y
x-y
x<sp>y
x.y
```

The <sp> stands for a space whenever needed to make it visible.

Since the period matches any single character, a way to deal with the "invisible" characters printed by l is available. For instance, if there is a line that when printed with the l command, appears as

. . . th\07is . . .

and it is desired to get rid of the "\07" (the bell character), the most obvious solution is to try

s/\07//

This will fail. The brute force solution, which is to retype the entire line, is a reasonable tactic if the line in question is not too long. However, for a very long line, retyping could result in additional errors. Since "\07" really represents a single character, the command

s/th.is/this/

gets the job done. The period matches the mysterious character between the "h" and the "i", whatever it is.

Since the period matches any single character, the command

s/./,/

converts the first character on a line into a ",".

As is true of many characters in **ed**, the period has several meanings depending on its context. This line shows all three:

.s/./.

- The first period is the line number of the line being edited, which is called "dot".
- The second period is a metacharacter that matches any single character on that line (in this instance the first character of the line).
- The third period is the only one that really is an honest literal period. On the right side of a substitution, the period is not special.

2.4.2 Backslash

Since a period means "any character", the question arises of what to do when a period is really needed. For example, to convert the line:

Now is the time.

into

Now is the time?

the backslash (\) is used. A backslash turns off any special meaning that the next character might have. In particular, \. converts the period from a "match anything" into a "match the period" statement. The \. pair of characters is considered by **ed** to be a single literal period. To replace the period with a question mark, the following command is used:

s/\./?/

The backslash can also be used when searching for lines that contain a special character. If a search is made to look for a line that contains

.PP

the search

```
/.PP/
```

is not adequate. It will find a line like

```
THE APPLICATION OF ...
```

The period matches the letter "A". But if the command

```
/\.PP/
```

is used, only the lines that contain ".PP" are found.

The backslash can also be used to turn off special meanings for characters other than the period. For example, to find a line that contains a backslash, the search

```
/\/
```

will not work because the \ is not a literal backslash, but instead means that the second / no longer delimits the search. A search can be made for a literal backslash by preceding a backslash with another \:

```
/\\/
```

Similarly, searches can be made for a slash (/) with

```
/\//
```

The backslash turns off the meaning of the immediately following / so that it does not terminate /.../ construction prematurely.

Some substitute commands, each of which will convert the line

```
\x\.\y
```

into the line

```
\x\y
```

are

```
s/\\\./
s/x../x/
s/..y/y/
```

The user's erase character and the line kill character (# and @ by default) must also be used with a backslash to turn off their special meaning. This is a feature of the UNIX operating system. When adding text with append (**a**), insert (**i**), or change (**c**) commands, the backslash is special only for the erase and line kill characters, and only one backslash should be used for each one needed.

2.4.3 Dollar Sign

In the left side of a substitute command or in a search command the dollar sign (**$**) stands for "the end of line". The word "time" is added to the end of the following phrase.

Now is the

with the following command:

s/$/<sp>time/

The result is

Now is the time

A space is needed before "time" in the substitute command, otherwise, the following will be printed.

Now is thetime

The second comma in the following line can be replaced with a period without altering the first.

Now is the time, for all good men,

The needed command is

s/,$/./

The $ provides context to indicate which specific comma. Without it the s command would operate on the first comma to produce

Now is the time. for all good men,

To convert

Now is the time.

into

Now is the time?

that was previously done with the backslash, the following command is used:

s/.$/?/

The $ has multiple meanings depending on context. In the line

$s/$/$/

- The first $ refers to the last line of the file.
- The second $ refers to the end of the line.
- The third $ is a literal dollar sign to be added to that line.

2.4.4 Circumflex

The circumflex (^), alias "hat" or "caret", stands for the beginning of the line. For example, if a search is made for a line that begins with "the", the command

/the/

will in all likelihood find several lines that contain "the" before arriving at the line that was wanted. But the command

```
/^the/
```

narrows the context, and thus arrives at the desired line more easily.

The other use of ^ is to enable context to be inserted at the beginning of a line.

```
s/^/<sp>/
```

places a space at the beginning of the current line.

Metacharacters can be combined. For example, to search for a line that contains only the characters

```
.PP
```

the command

```
/^\.PP$/
```

can be used.

2.4.5 Star

The star (*) is useful to replace all spaces between *x* and *y* with a single space, as in the following example:

text x y *text*

where *text* stands for lots of text, and there are an indeterminate number of spaces between *x* and *y*. The line is too long to retype, and there are too many spaces to count.

A regular expression (typically a single character) followed by a star stands for as many consecutive occurrences of that regular expression as possible. To refer to all the spaces at once, the following command is used:

```
s/x<sp>*y/x<sp>y/
```

The construction **<sp>*** means "as many spaces as possible". Thus **x<sp>*y** means: "an x, followed by as many spaces as possible, and then a y".

The star can be used with any character, not just space. If the original example was

text x--------y *text*

then all "-" characters can be replaced by a single space with the command

```
s/x-*y/x<sp>y/
```

If the original line was

text x.....y *text*

and if the following command was typed:

```
s/x.*y/x<sp>y/
```

what happens depends upon the occurrence of other x's or y's on the line. If there are no other x's or y's, then everything works, but it is blind luck, not good management. Since a period matches any single character, then .* matches as many single characters as possible. Unless the user is careful the star can eat up a lot more of the line than expected. If the line was

text x *text* x.....y *text* y *text*

then the command will take everything from the first "x" to the last "y", which, in this example, is undoubtedly more than wanted. The proper way is to turn off the special meaning of period with \.:

s/x\.*y/x<sp>y/

Now everything works since \.* means "as many periods as possible".

There are times when the pattern .* is exactly what is wanted. For example, to change

Now is the time for all good men ...

into

Now is the time.

the following deletes everything after the word "time":

s/<sp>for.*/./

There are a couple of additional pitfalls associated with * to be aware of. Most notable is that "as many as possible" means zero or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if this line contained

text xy *text* x y *text*

and the command is

s/x<sp>*y/x<sp>y/

the first "xy" matches this pattern, for it consists of an "x", zero spaces, and a "y". The result is that the substitute acts on the first "xy" and does not touch the later one that actually contains some intervening spaces.

The proper way is to specify a pattern like

/x<sp><sp>*y/

which says "an x, a space, as many more spaces as possible, and then a y" (in other words, one or more spaces).

The other startling behavior of * is also related to the zero being a legitimate number of occurrences of something followed by a star. The command

s/x*/y/g

when applied to the line

abcdef

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there is no "x" at the beginning of the line (so that gets converted into a "y"), nor between the "a" and the "b" (so that gets converted into a "y"), etc. The following command:

```
s/xx*/y/g
```

where "xx*" is "one or more x's", when applied to the line

```
abcdefxghi
```

produces

```
abcdefyghi
```

2.4.6 Brackets

Should a number that appears at the beginning of all lines of a file need to be deleted, a first thought might be to perform a series of commands like:

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
    .
    .
```

This is going to take forever if the numbers are long. Unless it is desired to repeat the commands over and over until finally all numbers are gone, the digits can be deleted on one pass. This is the purpose of brackets ([ ]).

The construction

```
[0123456789]
```

matches any single digit. The whole thing is called a "character class". With a character class, the job is easy. The pattern "[0123456789]*" matches zero or more digits (an entire number), so

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class; and just to confuse the issue, there are essentially no special characters inside the brackets. Even the backslash does not have a special meaning. The following command searches for special characters within the brackets:

```
/[.\$^[ ]/
```

Within a character class, the [is not special. To get a] into a character class, it should be placed as the first character in the class. For example:

```
/[ ].\$[ ]/
```

It is a nuisance to have to spell out the digits. They can be abbreviated as [0-9]; similarly, [a-z] stands for the lowercase letters and [A-Z] for uppercase letters.

The user can specify a character class that means "none of the following characters". This is done by beginning the class with a circumflex.

```
[^0-9]
```

which stands for "any character except a digit". The following search finds the first line that does not begin with a tab or space:

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. For example:

```
/^[^^]/
```

finds a line that does not begin with a circumflex.

2.4.7 Ampersand

The ampersand (&) is used primarily to save typing. For example, if the following is the original line:

```
Now is the time
```

and it needs to be

```
Now is the best time
```

the command

```
s/the/the best/
```

can be used, but it is unecessary to repeat the "the". The & is used to eliminate the repetition. On the right-hand side of a substitute command, the ampersand means "whatever was just matched", so in the command

```
s/the/& best/
```

the & represents "the". This is not much of a saving if the text matched is just "the"; but if it is something long or complicated or if it is something (such as .*) which matches a lot of text, the & can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length:

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side.

```
s/the/& best and & worst/
```

makes the original line

```
Now is the best and the worst time
```

and

```
s/.*/&? &!!/
```

converts the original line into

Now is the time? Now is the time!!

To get a literal ampersand, the backslash is used to turn off the special meaning.

s/ampersand/\&/

converts the word into the symbol. The **&** is not special on the left-hand side of a substitute, only on the right.

3. Operating On Lines

3.1 Substituting Newline Characters

The **ed** program provides a facility for splitting a single line into two or more lines by substituting in a newline character. If a line is unmanageably long because of editing or merely because of the way it is typed, it can be divided as follows:

text xy *text*

can be broken between the "x" and the "y" with the following substitute command:

```
s/xy/x\
y/
```

This is actually a single command although it is typed on two lines. Bearing in mind that \ turns off special meanings, it seems relatively intuitive that a \ at the end of a line would make the newline character there no longer special.

A single line can be made into several lines with this same mechanism. The word "very" in the following example can be put on a separate line preceded with the **nroff** formatter underline command (**.ul**):

text a very big *text*

The commands

```
s/<sp>very<sp>/\
.ul\
very\
/
```

convert the line into four shorter lines:

text a
.ul
very
big *text*

The word "very" is preceded by the line containing the ".ul" and spaces around "very" are eliminated at the same time.

When a new line is substituted in, dot is left pointing at the last line created.

3.2 Joining Lines

Lines may be joined together with the j command. Given the lines

```
Now is
<sp>the time
```

and if dot is set to the first line, then the j command joins them together. No spaces are added, which is why a space is shown at the beginning of the second line.

All by itself, a j command joins dot to dot+1. Any contiguous set of lines can be joined by specifying the starting and ending line numbers. For example:

```
1,$jp
```

joins all the lines into a big one and prints it.

3.3 Rearranging Lines

The & metacharacter stands for whatever was matched by the left side of an s command. Similarly, several pieces can be captured of what was matched; the only difference is it must be specified on the left side just what pieces the user is interested in. For instance, if there is a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

etc., and it was intended to have the initials to precede the name, as in:

```
A. B. Smith
C. Jones
```

it is possible to do this with a series of tedious and error-prone editing commands. The alternative is to "tag" the pieces of the pattern (in this case, the last name and the initials) and then rearrange the pieces. On the left side of a substitution if part of the pattern is enclosed between \(and \), whatever matched that part is remembered and available for use on the right side. On the right side, the symbol \1 refers to whatever matched the first \(...\) pair, \2 to the second \(...\) pair, etc.

The command

```
1,$s/^\([^,]*\),<sp>*\(.*\)/\2<sp>\1/
```

although hard to read, does the job. The first " \(...\) " matches the last name, which is any string up to the comma; this is referred to on the right side with "\1". The second " \(...\) " is whatever follows the comma and any spaces and is referred to as "\2".

With any complicated editing sequence, it is foolhardy to run it and hope. Global commands (see paragraphs 5.1 and 5.2) provide a way to print those lines affected by the substitute command.

4. Line Addressing in Editor

Line addressing in ed specifies the lines to be affected by editing commands. Previous constructions like

```
1,$s/x/y/
```

were used to specify a change on all lines. Most users are familiar with using a single newline character (or return) to print the next line and with

```
/string/
```

to find a line that contains "string". Less familiar is the use of

```
?string?
```

to scan backwards for the previous occurrence of "string". This is handy when the user realizes that the string to be operated on is back up the page (file) from the current line being edited.

The slash and question mark are the only characters that can be used to delimit a context search. Essentially, any character can be used as a delimiter in a substitute command.

4.1 Address Arithmetic

The next step is to combine the line numbers like . , $, /.../, and ?...? with + and −. Thus:

```
$−1
```

is a command to print the next to last line of the current file (i.e., one line before line $). For example:

```
$−5,$p
```

prints the last six lines. If there are not six lines, an error message will be indicated.

As another example:

```
.−3,.+3p
```

prints from three lines before the current line to three lines after, thus printing a bit of context. The + can be omitted:

```
.−3,.3p
```

is identical in meaning.

Another area in which to save typing effort in specifying lines is by using − and + as line numbers by themselves. For instance, a

```
−
```

by itself is a command to move back up one line in the file. Several minus signs can be strung together to move back up that many lines:

```
−−−
```

moves up three lines, as does "−3". Thus:

```
−3,+3p
```

is also identical to the examples above.

Since "−" is shorter than ".−1", constructions like

```
−,.s/bad/good/
```

are useful. This changes the first occurrence of "bad" to "good" on both the previous line and the current line.

The + and − can be used in combination with searches using /.../, ?...?, and $. The search

```
/string/--
```

finds the line containing "string" and positions dot two lines before it.

4.2 Repeated Searches

When the search command is

```
/horrible string/
```

and when the line is printed, it is discovered that it is not the horrible string that was wanted. It is necessary to repeat the search again, but it is not necessary to retype it. The construction

```
//
```

is a shorthand for "the string that was previously searched for", whatever it was. This can be repeated as many times as necessary. This also applies to the backwards search

```
??
```

which searches for the same string but in the reverse direction.

Not only can the search be repeated, but the // construction can be used on the left side of a substitute command to mean "the most recent pattern":

```
/horrible string/
    --- ed prints line with "horrible string"
s//good/p
```

To go backwards and change a line, the following command is used:

```
??s//good/
```

Of course, the & on the right-hand side of a substitute can still be used to stand for whatever got matched:

```
//s//&<sp>&/p
```

finds the next occurrence of whatever was searched for last, replaces it by two copies of itself, and then prints the line just to verify that it worked.

4.3 Default Line Numbers

One of the most effective ways to speed editing is by knowing which lines are affected by a command with no address and where dot will be positioned when a command finishes. Editing without specifying unnecessary line numbers can save a lot of typing. As the most obvious example, the search command

```
/string/
```

puts dot at the next line that contains "string". No address is required with commands like:

- s to make a substitution on the line

- **p** to print the line
- **l** to list the line
- **d** to delete the line
- **a** to append text after the line
- **c** to change the line
- **i** to insert text before the line.

If there was no "string", dot stays on the line where it was. This is also true if it was sitting on the only "string" when the command was issued. The same rules hold for searches that use **?...?**; the only difference is direction of search.

The delete command (**d**) leaves dot at the line following the last deleted line. However, dot points to the new last line when the last line is deleted.

Line-changing commands **a**, **c**, and **i** affect (by default) the current line if no line number is specified. They behave identically in one respect—after appending, changing, or inserting, dot points at the last line entered. For example, the following can be done without specifying any line number for the substitute command or for the second append command:

```
a
        --- text
        --- botch        (minor error)
.
s/botch/correct/        (fix botched line)
a
        --- more text
```

The following overwrites the major error and permits continuation of entering information:

```
a
        --- text
        --- horrible botch      (major error)
.
c
        --- fixed up line      (replace entire line)
        --- more text
```

The read command (**r**) will read a file into the text being edited, either at the end if no address is given or after the specified line if an address is given. In either case, dot points at the last line read in. The **0r** command can be used to read in a file at the beginning of the text, and the **0a** or **1i** commands can be used to start adding text at the beginning.

The write command (**w**) writes out the entire file. If the command is preceded by one line number, that line is written. Preceding the command by two line numbers causes a range of lines to be written. The **w** command does not change dot, therefore, the current line remains the same regardless of what lines are written. This is true even if a command like

```
/^\.AB/,/^\.AE/w abstract
```

is made, which involves a context search. Since the **w** command is easy to use, the text being edited should be saved regularly just in case the system crashes or a file being edited is clobbered.

The command with the least intuitive behavior is the **s** command. The dot remains at the last line that was changed. If there were no changes, then dot is unchanged. To illustrate, if there are three lines in the buffer and dot is sitting on the middle one

```
x1
x2
x3
```

the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command issued while dot pointed at the second line, then the result would be to change and print only the first line and that is where dot would be set.

4.4 Semicolon

Searches with /.../ and ?...? start at the current line and move forward or backward, respectively, until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like

```
.
.
.
ab
.
.
.
bc
.
.
.
```

Starting at line 1, one would expect that the command

```
/a/,/b/p
```

prints all the lines from the "ab" to the "bc", inclusive. This is not what happens. Both searches (for "a" and for "b") start from the same point, and thus they both find the line that contains "ab". The result is to print a single line. If there had been a line with a "b" in it before the "ab" line, then the print command would be in error since the second line number would be less than the first; and it is illegal to try to print lines in reverse order. This is because the comma separator for line numbers does not set dot while each address is processed. Each search starts from the same place.

In **ed**, the semicolon (;) can be used just like the comma with the single difference being that use of a semicolon forces dot to be set at that point while line numbers are being evaluated. In effect, the semicolon "moves" dot. Thus, in the example above, the command

```
/a/;/b/p
```

prints the range of lines from "ab" to "bc" because after the "a" is found dot is set to that line, and then "b" is searched for starting beyond that line. This property is most often useful in a very simple situation. If the need is to find the second occurrence of "string", then the commands

```
/string/
//
```

print the first occurrence as well as the second. The command

```
/string/;//
```

finds the first occurrence of "string" and sets dot there. Then it finds the second occurrence and prints only that line.

Searching for the second previous occurrence of "string", as in

```
?string?;??
```

is similar. Printing the third, fourth, etc. occurrence in either direction is left as an exercise.

When searching for the first occurrence of a character string in a file where dot is positioned at an arbitrary place within the file, the command

```
1;/string/
```

will fail if "string" occurs on line 1. It is possible to use the command

```
0;/string/
```

(one of the few places where 0 is a legal line number) to start the search at line 1.

4.5 Interrupting the Editor

If the user interrupts **ed** while performing a command (by depressing the BREAK key, the INTERRUPT key, or the user interrupt character [RUB OUT or DEL CHAR keys by default]), the file is put back together again. The file state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable. If the file is being read from or written into, substitutions are being made, or lines are being deleted, these will be stopped in some clean but unpredictable state in the middle of the command execution (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus, if a user interrupts **ed** while some printing is being done, dot is not sitting on the last printed line or even near it. Dot is returned to where it was when the **p** command was started.

5. Global Commands

5.1 Basic

Global commands (**g** and **v**) are used to perform one or more editing commands on all lines of a file. The **g** command operates on those lines that contain a specified string. As the simplest example, the command

```
g/THIS/p
```

prints all lines that contain the string "THIS". The string that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply. As another example:

```
g/^\./p
```

prints all lines that begin with a period.

The **v** command (there is no mnemonic significance to the letter "v") is identical to **g**, except that it operates on those lines that do not contain an occurrence of the string. So

```
v/^\./p
```

prints the lines that do not begin with a period.

The command that follows **g** or **v** can be almost any command. For example:

```
g/^\./d
```

deletes all lines that begin with a period, and

```
g/^$/d
```

deletes all empty (blank) lines.

Probably the most useful command that can follow a global command is the substitute command since this can be used to make a change and print each affected line for verification. For example, to change the word "This" to "THIS" everywhere in a file and verify that it really worked, the command is

```
g/This/s//THIS/gp
```

The use of // in the substitute command means "the previous pattern", in this case, "This". The **p** command is done on every line that matches the pattern, not just those on which a substitution took place.

Global commands operate by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** to use addresses, set dot, etc., quite freely. For example:

```
g/^\.PP/+
```

prints the line that follows each ".PP" command (the signal for a new paragraph in some formatting packages). The + means "one line past dot", and

```
g/topic/?^\.SH?1
```

searches for each line that contains "topic", scans backwards until it finds a line that begins ".SH" (a section heading) and prints the line that follows, thus showing the section headings under which "topic" is mentioned. Finally:

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with the ".EQ" and ".EN" formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

5.2 Multiline

It is possible to do more than one command under the control of a global command although the syntax for expressing the operation is not especially natural or easy. As an example, suppose the task is to change "x" to "y" and "a" to "b" on all lines that contain "string". Then:

```
g/string/s/x/y/\
s/a/b/
```

is sufficient. The backslash signals the **g** command that the set of commands continues on the next line. It terminates on the first line that does not end with \. A substitute command can not be used to insert a newline character within a **g** command.

The command

```
g/x/s//y/\
s/a/b/
```

does not work as expected. The remembered pattern is the last pattern that was actually executed, so sometimes it will be "x" (as expected) and sometimes it will be "a" (not expected). The desired pattern should be spelled out:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a, c, and i** commands (append, change, and insert) under a global command. As with other multiline constructions, all that is needed is to add a \ at the end of each line except the last. Thus to add a **.nf** and **.sp** command before each ".EQ" line, the following is typed:

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a period to terminate the **i** command unless there are further commands being done under the global. On the other hand, it does no harm to put it in.

It is good practice, after each global command, to check that the command did only what was desired. Surprises sometimes happen. When they do occur, the **u** command (undo) is useful to negate what was done by the last command.

6. Cut and Paste

One editing area in which nonprogrammers do not seem confident is the "cut and paste" operations. There are two areas in which the operations can be performed. Using the UNIX operating system command functions, the following can be done:

- Changing the name of a file
- Making a copy of a file somewhere else
- Combining files
- Removing a file.

The text editor (**ed**) function performs the following operations.

- Inserting one file in the middle of another
- Splitting a file into pieces
- Moving a few lines from one place to another in a file
- Copying lines.

Most of these operations are actually quite easy if the task is defined and precautions are taken when entering the commands.

6.1 Command Functions

Changing file names, making copies of files, combining files, and removing files are handled with the UNIX operating system commands.

6.1.1 Changing Name of Files

If there is a file named *oldname* and if it needs to be renamed to *newname*, the move command (**mv**) will do the job. It moves the file from one name to another (the target file), for example

```
mv oldname newname
```

Note: If there is already a file with the new name, its contents will be overwritten with information from the other (*oldname*) file. The one exception is that a file cannot be moved to itself; therefore, the following command is illegal.

```
mv oldname oldname
```

6.1.2 Copying Files

Sometimes a copy of a file is needed while retaining the original file. This might be because a file needs to be worked on and yet have a back-up in case something happens to the file. In any case, the copy is made with the copy command (**cp**). To make a copy of a file named *good*, the following command will place a copy in a file named *savegood*:

cp *good savegood*

Two identical copies of the file *good* exist. If *savegood* previously contained something, it is overwritten.

To get the file *savegood* back to its original filename, *good*, the following commands are used:

mv *savegood good*

if *savegood* is not needed anymore or

cp *savegood good*

to retain a copy of *savegood*.

In summary, **mv** renames a file; **cp** makes a duplicate copy. Both commands overwrite the target file if one already exists unless write permission is denied by the mode of the file.

6.1.3 Combining Files

A familiar requirement is that of collecting two or more files into one big file, *bigfile*. This is needed, for example, when the author of a paper decides that several sections are to be combined. There are several ways to do this; the cleanest is a command called **cat** (not all commands have 2-letter names). The word **cat** is short for "concatenate", which is exactly what is desired. The command

cat *file*

prints the contents of the *file* on the terminal. The command

cat *file1 file2*

causes the contents of *file1* and *file2* to be printed on the terminal, in that order, but does not place them in *bigfile*.

There is a way to tell the system to put the same information in a file instead of printing on the terminal. The way to do it is to add to the command line the > character and the name of the file where the output is to go. The command

cat *file1 file2* > *bigfile*

is used and the job is done. As with **cp** and **mv**, when something is put into *bigfile*, anything already there is destroyed. The ability to capture the output of a program can be used with any command that prints on a terminal. Several files can be combined, not just two.

cat *file1 file2 file3 ...* > *bigfile*

collects many individual files.

Sometimes a file needs to be appended to the end of another file. For example:

cat *good good1* > *temp*
mv *temp good*

is the most direct way. The following command:

cat *good good1* > *good*

does not work because the > empties *good* before the **cat** program begins. The easiest way is to use a variant of >, called >>. In fact, >> is identical to > except that instead of clobbering the old file it adds something to the end. Thus the command

cat *good1* >> *good*

adds *good1* to the end of *good*. If *good* does not exist, this makes a copy of *good1* called *good*.

6.1.4 Removing Files

If a file is not needed, it can be removed. The **rm** command

rm *savegood*

irrevocably deletes the file called *savegood* if the user had write permission.

6.2 Text Editor Functions

Manipulating pieces of files, individual lines, or groups of lines are handled with the text editor.

6.2.1 File Names

It is important to know the editor (**ed**) commands for reading and writing files. Equally useful is the edit command (**e**). Within **ed**, the command

e *newfile*

says "edit a new file called *newfile* without leaving the text editor". The **e** command discards whatever is being worked on and starts over on *newfile*. This is the same as if one had quit with the **q** command and reentered **ed** with a new file name except that if a pattern has been remembered, a command like // will still work.

When entering **ed** with the command

ed *file*

ed remembers the name of the file, and any subsequent **e**, **r**, or **w** commands that do not contain a file name will refer to this remembered file. Thus:

```
ed file1
    ---     (editing)
w           (writes back in file1)
e file2     (edit different file, without leaving ed)
    ---     (editing on file2)
w           (writes back on file2)
```

etc., does a series of edits on various files without leaving **ed** and without typing the name of any file more than once. By examining the sequence of commands in this example, it can be seen why many operating systems use **e** as a synonym for **ed**.

The current file name can be found at any time with the **f** command by typing **f** without a file name. Also, the name of a remembered file can be changed with **f**. A useful sequence is

```
ed precious
f junk
    ---     (editing)
```

This obtains a copy of the file **precious** and guarantees that a subsequent **w** command without a filename will write to *junk* and will not overwrite the original file.

6.2.2 Inserting One File Into Another

When a file is to be inserted into another, the **r** command can be used. For example, if the file *table* is to be inserted just after the reference to "Table 1", the following can be used:

```
/Table 1/
Table 1 shows that...     (response from ed)
.r table
```

The critical line is the last one. The **.r** command reads a file in after dot. An **r** command without any address adds lines to the end of the file, so it is equivalent to the **$r** command.

6.2.3 Writing Out Part of a File

Another feature is writing to another file part of the document that is being edited. For example, it is possible to split into a separate file the table from the previous example, so it can be formatted or tested separately. If in the file being edited, there is

```
        --- text
.TS
        --- lots of stuff
.TE
        --- text
```

(which is the way a table is set up (as explained in Section III) to isolate the table in a separate file called *table*, first the start of the table (the .TS line) is found, and then the interesting part is written on file *table:*

```
/^\.TS/
.TS        (response from ed)
.,/^\.TE/w table
```

The same job can be accomplished with the single command

```
/^\.TS/;/^\.TE/w table
```

The point is that the **w** command can write out a group of lines instead of the whole file. A single line can be written by using one line number instead of two. For example, if a complicated line was just typed and it will be needed again, it should be saved and read in later rather than retyped:

```
a
        --- lots of stuff
        --- stuff to repeat
.
.w temp
a
        --- more stuff
.
.r temp
a
        --- more stuff
.
```

6.2.4 Moving Lines Around

Moving a paragraph from its present position in a paper to the end can be done several ways. For example, it is assumed that each paragraph in the paper begins with the formatting command ".PP". The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, and then read in the temporary file at the end. If dot is at the ".PP" command that begins the paragraph, this is the sequence of commands:

```
.,/^\.PP/-w temp
.,//-d
$r temp
```

This states that from where dot is now until one line before the next ".PP" write onto file *temp*. The same lines are deleted and the file *temp* is read in at the end of the working file.

An easier way is to use the move command (**m**) that **ed** provides. This does the whole set of operations at one time without a temporary file. The **m** command is like many other **ed** commands in that it takes up to two line numbers in front to tell which lines are to be affected. It is also followed by a line number that tells where the lines are to go. Thus:

```
line1,line2m line3
```

says "move all the lines from **line1** through **line2** to after **line3**". Any of "line1", etc., can be patterns between slashes, dollar signs, or other ways to specify lines. If dot is at the first line of the paragraph, the command

```
.,/^\.PP/-m$
```

will also accomplish this task.

As another example of a frequent operation, the order of two adjacent lines can be reversed by moving the first one after the second. If dot is positioned at the first line, then

```
m+
```

does it. It says to move the line to after the dot. If dot is positioned on the second line:

```
m--
```

does the interchange.

The **m** command is more concise and direct than writing, deleting, and rereading. The main difficulty with the **m** command is that if patterns are used to specify both the line being moved and the target line, they must be specified properly or the wrong lines may be moved. The result of a botched **m** command can be a costly mistake. Doing the job a step at a time makes it easier to verify that each step accomplished what was wanted. It is also a good idea to issue a **w** command before doing anything complicated; then if an error is made, it is easy to back up.

6.2.5 Copying Lines

The **ed** program provides a transfer command (**t**)for making a copy of a group of one or more lines at any point. This is often easier than writing and reading. The **t** command is identical to the **m** command except instead of moving lines it duplicates them at the place referenced. Thus:

```
1,$t$
```

duplicates the entire contents that is being edited. A more common use for **t** is creating a series of lines that differ only slightly. For example:

```
a
	--- long line of stuff
.
t.		(make a copy)
s/x/y/		(change it a bit)
t.		(make third copy)
s/y/z/		(change it a bit)
```

6.2.6 Marks

The **ed** program provides for marking a line with a particular name so that the line can be referenced later by its name regardless of its line number. This can be useful for moving lines and for keeping track of them as they move. The mark command is **k**. The mark name must be a single lowercase letter. The command

kx

marks the current line with the name "x". If a line number precedes the **k**, that line is marked. The marked line can then be referred to with the address

'x

Marks are most useful for moving things around. The first line of the block to be moved is found and marked with **ka**. Then the last line is found and marked with **kb**. Dot is then positioned at the place where the lines are to go and the following command is performed:

'a,'bm.

Note: Only one line can have a particular mark name associated with it at any given time.

6.3 Temporary Escape

Sometimes it is convenient to temporarily escape from the text editor to do some UNIX operating system command without leaving the text editor. The escape command (!) provides a way to do this. If the command

!<any UNIX operating system command>

is entered, the current editing state is suspended; and the command asked for is executed. When the command finishes, **ed** will return a signal by printing another ! and editing can be resumed.

Any UNIX operating system command may be performed including another **ed** (this is quite common). In this case, another ! can be done.

7. Supporting Tools

There are several related tools and techniques which are relatively easy to learn after **ed** has been learned because they are based on **ed**. This section gives some cursory examples of these tools, more to indicate their existence than to provide a complete tutorial.

7.1 Global Printing From a Set of Files (grep)

Sometimes all occurrences of some word or pattern in a set of files need to be found in order to edit them or perhaps to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest. If there are many files, this can be tedious; and if the files are really big, it may be impossible because of limits in **ed**.

The **grep** program was written to get around these limitations. Search patterns described in this section are often called "regular expressions", and "grep" stands for

g/re/p

This describes what **grep** does—it prints every line in a set of files that contains a particular pattern. Thus:

grep 'string' file1 file2 file3 ...

finds "string" wherever it occurs in any of the files *file1*, *file2*, etc. The **grep** program also indicates the file in which the line was found, so it can be edited later if needed.

The pattern represented by "string" can be any pattern that can be used in the text editor since **grep** and **ed** use the same mechanism for pattern searching. It is wisest to enclose the pattern in single quotes ('...') if

it contains any nonalphabetic characters since many such characters also mean something special to the UNIX operating system command interpreter (the "shell"). Without single quotes, the command interpreter will try to interpret them before **grep** has the opportunity.

There is also a way to find lines that do not contain a pattern:

```
grep -v 'string' file1 file2 ...
```

finds all lines that do not contain "string". The **-v** must occur in the position shown. Given **grep** and **grep -v**, it is possible to select all lines that contain some combination of patterns. For example, to obtain all lines that contain "x" but not "y":

```
grep x file... | grep -v y
```

The pipe notation (|) causes the output of the first command to be used as input to the second command.

7.2 Editing Scripts

If a fairly complicated set of editing operations is to be performed on an entire set of files, the easiest thing to do is to make a script file, i.e., a file that contains the operations to be performed and then apply this script to each file in turn. For example, if every instance of "This" needs to be changed to "THIS" and every instance of "That" needs to be changed to "THAT" in a large number of files, a file *script* is made with the following contents:

```
g/This/s//THIS/g
g/That/s//THAT/g
w
q
```

The following is done:

```
ed file1 <script
ed file2 <script
...
```

This causes **ed** to take its commands from the prepared script. The whole job has to be planned in advance.

By using the UNIX operating system command interpreter [**sh**(1)], a set of files can be cycled automatically with varying degrees of ease.

STREAM EDITOR

1. Introduction

The stream editor (**sed**) is a noninteractive context editor that runs on the UNIX operating system. The **sed** software is designed to be especially useful in the following cases:

- When editing files too large for comfortable interactive editing
- When editing any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode
- When performing multiple global editing functions efficiently in one pass through the input file.

Because only a few lines of the input file reside in memory at one time and no temporary files are used, the effective size of a file that can be edited is limited only by the requirement that the input and output files fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to the **sed** program as a command file. For complex edits, this saves considerable typing and attendant errors. The **sed** program running from a command file is much more efficient than an interactive editor even if that editor can be driven by a prewritten script.

The principal loss of functions, if compared to an interactive editor, are lack of relative addressing (because of the line-at-a-time operation) and the lack of immediate verification that a command has done what was intended.

The **sed** program is a lineal descendant of the text editor, **ed**. Because of the differences between interactive and noninteractive operations, considerable changes have been made between **ed** and **sed**.

2. Overall Operation

The **sed** program by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line. (See Command Line Flags, paragraph 2.1.)

The general format of an editing command is

[address1,address2] function [arguments]

One or both addresses may be omitted. Any number of blanks or tabs may separate the addresses from the function. The function must be present. Arguments may be required or optional according to the function given. Tab characters and spaces at the beginning of lines are ignored.

2.1 Command Line Flags

Three flags are recognized on the command line:

−n	tells the **sed** program not to copy all lines, but only those specified by **p** (print) functions or **p** flags after **s** (substitute) functions
−e	tells the **sed** program to take the next argument as an editing command
−f	tells the **sed** program to take the next argument as a file name; the file should contain editing commands—one to a line.

2.2 Order of Application of Editing Commands

Before any input file is opened, all editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file).

- Commands are compiled in the order encountered; generally, the order they will be attempted at execution time.
- Commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the **t** (test substitution) and **b** (branch) flow-of-control commands. When the order of application is changed by these commands, it remains true that the input line to any command is the output of any previously applied command.

2.3 Pattern Space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the next command (**N**).

2.4 Examples

Examples scattered throughout the following paragraphs use the following standard input text, except where noted:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

The command

```
2q
```

will copy the first two lines of the input and quit. The output will be

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

3. Selecting Lines for Editing

Input file lines that editing commands are to be applied to can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address pair) by grouping commands with curly braces ({ }).

3.1 Line Number Addresses

A line number is a decimal integer. As each line is read from the input, a line number counter is incremented. A line number address matches (selects) the input line causing the internal counter to equal the address line number. The counter runs cumulatively through multiple input files. It is not reset when a new input file is opened. As a special case, the **$** character matches the last line of the last input file.

3.2 Context Addresses

A context address is a pattern (a regular expression) enclosed in slashes (/.../). Regular expressions recognized by the **sed** program are constructed as follows:

- An ordinary character is a regular expression and matches that character.
- A circumflex (^) at the beginning of a regular expression matches the null character at the beginning of a line.
- A dollar sign ($) at the end of a regular expression matches the null character at the end of a line.
- The characters (\n) match an embedded newline character but not the newline character at the end of the pattern space.
- A period (.), sometimes called dot, matches any character except the terminal newline character of the pattern space.
- A regular expression followed by an asterisk (*) matches any number (including 0) of adjacent occurrences of the regular expression it follows.
- A string of characters in square brackets ([ ]) matches any character in the string and no others. If, however, the first character of the string is a circumflex (^), the regular expression matches any character except the characters in the string and the terminal newline character of the pattern space. The circumflex is the only metacharacter recognized within the square brackets. If] needs to be in the set of square brackets, it should be the first nonmetacharacter. For example:

  ```
  [ ]...]      Includes ]
  [^]...]      Does not include ]
  ```

- A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
- A regular expression between the sequences \(and \) is identical in effect to the unadorned regular expression but has side effects which are described under the **s** command (substitute function) below.
- The expression \d means the same string of characters matched by an expression enclosed in \(and \) earlier in the same pattern. The *d* is a single digit; the string specified is that beginning with occurrence *d* of \(counting from the left. For example, the following expression matches a line beginning with two repeated occurrences of the same string:

  ```
  ^\(.*\)\1
  ```

- The null regular expression standing alone (e.g., //) is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal character (to match an occurrence of itself in the input), the special character is preceded by a backslash (\).

For a context address to match, the input requires that the whole pattern within the address match some portion of the pattern space.

3.3 Number of Addresses

Commands in the following paragraphs can have 0, 1, or 2 addresses. Under each command, the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has *no addresses*, it is applied to every line in the input.

If a command has *one address*, it is applied to all lines which match that address.

If a command has *two addresses*, it is applied to the first line which matches the first address and to all subsequent lines until (and including) the first subsequent line which matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated. Two addresses are separated by a comma. Some examples are:

/an/	matches lines 1, 3, and 4 in the sample text
/an.*an/	matches line 1
/^an/	matches no lines
/./	matches all lines
/\./	matches line 5
/r*an/	matches lines 1, 3, and 4 (number = 0)
/\(an\).*\1/	matches line 1.

4. Functions

Functions are named by a single alphabetic character. In the following function summaries, the maximum number of allowable addresses is enclosed in parentheses, followed by the single character function name. Possible arguments are enclosed in angle brackets (< >), and a description of each function is given. Angle brackets around arguments are not part of the argument and should not be typed in actual editing commands.

4.1 Whole Line Oriented Functions Summary

(2)d

The **d** function deletes from the file (does not write to the output) those lines matched by its addresses. It also has the side effect that no further commands are attempted on the corpse of a deleted line. As soon as the **d** function is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line.

(2)n

The **n** function reads the next line from the input, replacing the current line, and the current line is written to the output. The list of editing commands is continued following the **n** command.

(1)a\
<text>

The **a** function causes the argument <text> to be written to the output after the line matched by its address. The **a** command is inherently multiline; **a** must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, interior newline characters must be hidden by a backslash character (\) immediately preceding the newline character. The <text> is terminated by the first unhidden

newline character not immediately preceded by a backslash. Once an **a** function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. Even if that line is deleted, <text> will still be written to the output. The <text> is not scanned for address matches, and no editing commands are attempted on it. The **a** function does not cause a change in the line number counter.

(1)i\
<text>

The **i** function behaves identically to the **a** function except that <text> is written to the output before the matched line. All other comments about the **a** function apply to the **i** function.

(2)c\
<text>

The **c** function deletes lines selected by its addresses and replaces them with the lines in <text>. Like **a** and **i**, **c** must be followed by a newline character hidden by a backslash; interior newline characters in <text> must be hidden by backslashes. The **c** command may have two addresses, and therefore select a range of lines. If it does, all lines in the range are deleted, but only one copy of <text> is written to the output, not one copy per line deleted. As with **a** and **i**, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line number counter. After a line has been deleted by a **c** function, no further commands are attempted on the corpse. If text is appended after a line by **a** or **r** functions and the line is subsequently changed, the text inserted by the **c** function will be placed before the text of the **a** or **r** functions (the **r** function is described later).

For text put in the output by these functions, leading blanks and tabs will disappear as in **sed** commands. To get leading blanks and tabs into the output, the first desired blank or tab is preceded by a backslash. The backslash will not appear in the output. The list of editing commands for example:

```
n
a\
XXXX
d
```

applied to the standard input produces

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n
i\
XXXX
d
```

or

```
n
c\
XXXX
```

4.2 Substitute Function

One important substitute function that changes parts of lines selected by a context search within the line is

(2)s<pattern><replacement><flags>

The **s** function replaces the part of a line selected by <pattern> with <replacement>. It can be read

Substitute for <pattern>, <replacement>

4.2.1 Pattern

The <pattern> argument contains a pattern exactly like the patterns in addresses. The only difference between <pattern> and a context address is that the context address must be delimited by slash (/) characters; <pattern> may be delimited by any character other than space or newline. By default, only the first string matched by <pattern> is replaced unless the **g** flag (below) is invoked.

4.2.2 Replacement

The <replacement> argument begins immediately after the second delimiting character of <pattern> and must be followed immediately by another instance of the delimiting character (thus there are exactly three instances of the delimiting character). The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

\&	is replaced by the string matched by <pattern>.
\d	is replaced by substring **d** (*d* is a single digit), matched by parts of <pattern>, and enclosed in \(and \). If nested substrings occur in <pattern>, substring **d** is determined by counting opening delimiters (\(). As in patterns, special characters may be made literal characters by preceding them with backslash (\).

4.2.3 Flags

The <flags> argument may contain the following:

g	Substitute <replacement> for all nonoverlapping instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters. Characters put into the line from <replacement> are not rescanned.
p	Print the line if a successful replacement was done. The **p** flag causes the line to be written to the output if and only if a substitution was actually made by the **s** function. If several **s** functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line will be written to the output—one for each successful substitution.
w <filename>	Write the line to a file if a successful replacement was done. The **w** flag causes lines which are actually substituted by the **s** function to be written to a file named by <filename>. If

<filename> exists before **sed** is run, it is overwritten; if not, it is created. A single space must separate **w** and <filename>. The possibilities of multiple, somewhat different copies of one input line being written are the same as for **p**. A maximum of ten different file names may be mentioned after **w** flags and **w** functions.

4.2.4 Examples

The command

```
s/to/by/w changes
```

applied to the standard input produces on the output

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and on the file *changes*

```
Through caverns measureless by man
Down by a sunless sea.
```

If the no-copy option is in effect, the command

```
s/[.,;?:]/*P&*/gp
```

produces

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

To illustrate the effect of the **g** flag, the command

```
/X/s/an/AN/p
```

produces (assuming no-copy mode)

```
In XANadu did Kubla Khan
```

and the command

```
/X/s/an/AN/gp
```

produces

```
In XANadu did Kubla KhAN
```

4.3 Input/Output Functions

(2)**p** The *print* function writes addressed lines to the standard output file. They are written at the time the **p** function is encountered regardless of what succeeding editing commands may do to the lines.

(2)**w** <filename> The *write* function writes addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line regardless of what subsequent editing commands may do to them. Exactly one space must separate the **w** and <filename>. A maximum of ten different files may be mentioned in write functions and **w** flags after **s** functions combined.

(1)**r** <filename> The *read* function reads the contents of <filename> and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If **r** and **a** functions are executed on the same line, the text from **a** functions and **r** functions is written to the output in the order that the functions are executed. Exactly one space must separate the **r** and <filename>. If a file mentioned by an **r** function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

Note: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in **w** functions or flags. That number is reduced by one if any **r** functions are present (only one read file is opened at a time).

If the file *note1* has the following contents

```
Note: Kubla Khan (more properly Kublai Khan;
1216-1294) was the grandson and most eminent
successor of Genghiz (Chingiz) Khan and founder
of the Mongol dynasty in China.
```

then the command

```
/Kubla/r note1
```

produces

```
In Xanadu did Kubla Khan
    Note: Kubla Khan (more properly Kublai Khan;
    1216-1294) was the grandson and most eminent
    successor of Genghiz (Chingiz) Khan and founder
    of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

4.4 Multiple Input Line Functions

Three functions, all spelled with capital letters, deal with pattern spaces containing embedded newline characters. They are intended principally to provide pattern matches across lines in the input.

(2)**N** The next input line is appended to the current line in the pattern space. The two input lines are separated by an embedded newline character. Pattern matches may extend across embedded newline characters.

(2)D Delete first part of the pattern space. Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline character was the terminal newline character), read another line from the input. In any case, begin the list of editing commands again from the beginning.

(2)P Print first part of the pattern space. Print up to and including the first newline character in the pattern space.

The **P** and **D** functions are equivalent to their lowercase counterparts if there are no embedded newline characters in the pattern space.

4.5 Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)**h** Hold pattern space.The **h** function copies contents of the pattern space into a hold area destroying previous contents.

(2)**H** Hold pattern space. The **H** function appends contents of the pattern space to contents of the hold area. Former and new contents are separated by a newline character.

(2)**g** Get contents of hold area. The **g** function copies contents of the hold area into the pattern space destroying previous contents.

(2)**G** Get contents of hold area. The **G** function appends contents of the hold area to contents of the pattern space. Former and new contents are separated by a newline character.

(2)**x** Exchange. The exchange command interchanges contents of the pattern space and the hold area.

The following are examples:

```
1h
1s/did.*//
1x
G
s/\n/   :/
```

when applied to the standard input text, produce

```
In Xanadu did Kubla Khan    :In Xanadu
A stately pleasure dome decree:    :In Xanadu
Where Alph, the sacred river, ran    :In Xanadu
Through caverns measureless to man    :In Xanadu
Down to a sunless sea.    :In Xanadu
```

4.6 Flow of Control Functions

These functions do no editing on the input lines but control the application of functions to the lines selected by the address part.

(2)! Don't
The don't command causes the next command (written on the same line) to be applied to those input lines not selected by the address part.

(2){ Grouping
The grouping command causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the { or on the next line. The group of commands is terminated by a matching } standing on a line by itself. Groups can be nested.

(0):<label> Place a label
The label function marks a place in the list of editing commands which may be referred to by **b** and **t** functions. The <label> may be any sequence of eight or fewer characters. If two different colon functions have identical labels, a compile time diagnostic will be generated; and no execution attempted.

(2)b<label> Branch to label
The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all editing commands have been compiled, a compile time diagnostic is produced; and no execution is attempted. A **b** function with no <label> is a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

(2)t<label> Test substitutions
The **t** function tests whether any successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by reading a new input line and executing a **t** function.

4.7 Miscellaneous Functions

(1)= The = function writes to standard output the line number of the line matched by its address.

(1)q The **q** function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

MISCELLANEOUS FACILITIES

Several miscellaneous facilities exist (via UNIX operating system commands) to aid in the development of documentation. These facilities are easy to access and are very effective. Their use is beneficial in documentation development. Some available miscellaneous facilities are described briefly in the following list. The User's Manual—UNIX Operating System has a more detailed description.

bdiff The **bdiff** facility is used in a manner analogous to **diff** to find which lines must be changed in two files to bring them into agreement. Its purpose is to allow processing of files which are too large for **diff**.

cat The **cat** facility reads each file in sequence and writes it on the standard output. Thus:

cat *file*

prints the file named *file*, and

cat *file1 file2*>*file3*

concatenates *file1* and *file2* and places the result in *file3*.

cmp The **cmp** facility compares two files. Under default options, **cmp** makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred.

comm The **comm** facility selects or rejects lines common to two sorted files. It reads *file1* and *file2* and produces a 3-column output as follows: lines only in *file1*, lines only in *file2*, and lines in both files.

diff The **diff** facility is a differential file comparator. It tells what lines must be changed in two files to bring them into agreement.

diff3 The **diff3** facility is a 3-way differential file (files up to 64K) comparator. It compares three versions of a file and publishes disagreeing ranges of text flagged with special codes.

diffmk The **diffmk** facility marks the differences between files. It compares two versions of a file and creates a third file that includes "change mark" commands for the **nroff** or **troff** formatter.

grep Commands of the **grep** facility search the input files for lines matching a pattern. Normally, each line found is copied to the standard output. The **grep** patterns are limited regular expressions in the style of **ed**. The **egrep** patterns are full regular expressions. The **fgrep** patterns are fixed strings.

pr The **pr** facility prints the named files on the standard output. If file is — or if no files are specified, the standard input is assumed.

sdiff The **sdiff** facility uses the output of **diff**(1) to produce a side-by-side listing of two files indicating those lines that are different. Each line of the two files are printed with a blank gutter between them if the lines are identical, a > in the gutter if the line exists only in *file1*, a < in the gutter if the line exists only in *file2*, and a ! for lines that are different.

sort The **sort** facility sorts lines of all the named files together and writes the results on the standard output.

spell

The **spell** facility collects words from the named files and looks them up in a spelling list. Words that do not occur in the spelling list nor can be derived from them are printed on the standard output. The **spellin** and **spellout** are two additional subroutines of **spell**.

split

The **split** facility splits a file into pieces.

typo

The **typo** facility searches through a document for unusual words, typographical errors, and hapax legomena and prints them on the standard output.

uniq

The **uniq** facility reports repeated lines in a file. It reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file.